An approach for constructing a domain definition metamodel with ATL

Vanea Chiprianov, Yvon Kermarrec, Patrick Alff

An Approach for Constructing a Domain Definition Metamodel with ATL

Vanea Chiprianov[1,3], Yvon Kermarrec[1,3] and Patrick D. Alff[2]

[1] Institut Telecom, Telecom Bretagne, UMR CNRS 3192 Lab-STICC
Technopôle Brest Iroise, CS 83818 29238 Brest Cedex 3, France
{vanea.chiprianov},{yvon.kermarrec}@telecom-bretagne.eu
[2] BT-North America, 2160 E. Grand Ave, El Segundo, CA 90245, United States
[3] Universite europeenne de Bretagne, France

Abstract. Present day Telecommunications competitive market requires a rapid definition process of new services. To ensure this, we propose to replace the current paper-based process with a computer-aided one. Central to this later process is an information model that captures domain specific knowledge. We approach its construction by defining model querying and model transformation rules in ATL over existing network abstraction layers. We also report on the way we used ATL to define these rules and the benefits of doing so, and pinpoint issues that may be addressed in future ATL releases.

1 Introduction

Present day Telecommunications customer-centric market is characterized by a high demand rate for new services and fierce competition. To remain competitive, service providers need to speed up their service definition process by shortening the concept-to-market time and designing the product right-the-first-time. Currently, this is a paper-based process, relying mainly on trays of documents being exchanged between service designers and programmers. We propose to replace this process with a computer-aided one, which will iteratively capture domain specific knowledge in a Domain Definition Metamodel (DDMM) (the central entity in Fig. 1; in this figure we represent with filled ellipses what we have already done; in parentheses we indicate the toolkit we used). A DDMM provides a sharable, capitalizable, stable and organized structure of information.

Starting from the DDMM, we can define one or several Domain Specific Languages (DSLs) (Sect. 2) which increase the performance of service designers. The DDMM can support the service designers in their collaborative work when defining a new service. It can also be used for verifying properties on models that were defined using the aforementioned DSL. Therefore, as highlighted by [1] also, **the DDMM is central** to our approach. It is essential that it is wide enough to provide the majority of the concepts service designers need, but also that it is formal and close enough to existing Network Abstraction Layers (NALs) to enable mapping of service specific concepts with network specific components.

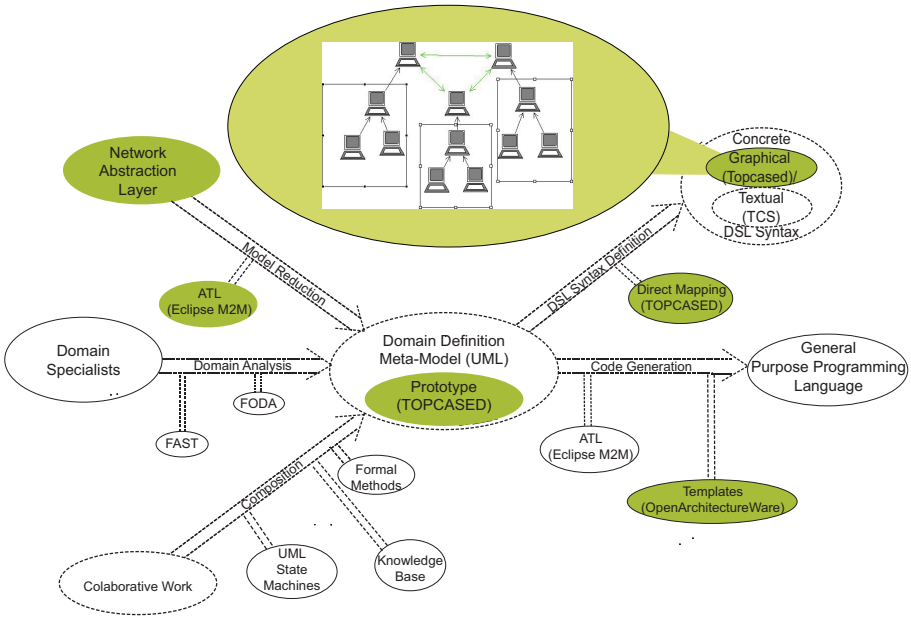

Fig. 1. Telecommunications-specific Modeling

Consequently, we decided to construct the DDMM by simplifying existing NALs (Sect. 4) and it constitutes the main focus of this document. The approach of extracting only the relevant aspects of an information model for the generation of DSLs has already been used, for example by [2]. However, their method consists in tagging the information model, whereas we construct a new model by applying transformation rules.

An NAL captures a lot of information, among which there is a big part of the service domain, but in a much more detailed manner than necessary for a designer (an NAL can have tens of thousands of entities, while the service domain has hundreds of concepts). This introduces an additional complexity. The solution is to present the users only the information that is pertinent for them. By eliminating most of the entities that are unknown to service designers and shrinking the inheritance hierarchies, we elaborate such a model. Some NALs are expressed as large class models in UML, so we use model querying (Sect. 3) and transformation techniques and write the model querying and transformation rules using ATL [3] (Sect. 4.2). Because we define more than queries on the input model, changing relations, we go beyond model querying, into model transformation.

2 A Simple Graphical Telecommunications Specific Modeling Language (SGTSML)

If we consider the DDMM corresponding to the **abstract syntax** of a language (for more details of language definition using model-based tools see [4]), we can define a modeling language for service definition and generate tools for it (e.g.; graphical or textual editor). Moreover, if needed to describe disparate aspects of the service (e.g.; structural, behavioral), the DDMM can be augmented with the information needed to define several DSLs. Sharing the DDMM between several DSLs ensures a consistent view. A similar approach has already been proposed by [2] for policy specification.

We approached the definition of the DDMM with an iterative method in mind, so we started with a simple prototype. This prototype is aimed at defining a simple virtual private network (Fig. 2). The prototype consists of a *Network*, which may contain several inner networks and several *Node*s. The nodes are either *Computer*s, *Internet* or *Router*s; they are connected by links which constitute outlinks for the source nodes, and inlinks for the target nodes. The routers can be either customer edge routers (*CE*) or provider edge routers (*PE*). Each PE and CE has an *Interface*, which contains a virtual routing and forwarding (*VRF*) table containing the VrfRouteTargets and information about the neighboring PEs (*BgpIpv4AddressFamilyNeighbor*s). PEs use the Border Gateway Protocol (*BgpRoutingProtocol*) to communicate with each other. We also enriched the DDMM with validation rules [5], thus enabling domain level validation. As tool for defining the DDMM we chose TOPCASED [6], a strongly model oriented system engineering toolkit for critical and embedded applications.

For the **concrete syntax** (see filled ellipses on the right top of Fig. 1) we considered that a graphical syntax would be much easier to use by service designers, as it provides a synthetic, high-level view of the system being considered. Therefore, we defined one using TOPCASED, which has a feature that allows automatic generation of graphical editors for DSLs based on their Metamodel (MM). Using TOPCASED, we also generated the graphical editor for SGTSML.

To describe the **semantics** of our SGTSML we decided to use the semantics of an existing general purpose object-oriented programming language, Smalltalk. Consequently, we defined template-based code generation rules towards Smalltalk, using OpenArchitectureWare [7]. More details about the definition of SGTSML can be found in [5].

3 Model Querying and Transformation

UML class models can quickly become very large, comprising thousands of classes. Viewing the entire model imposes a high cognitive charge on designers. They usually need to concentrate only on the classes related to a precise functionality (i.e.; a model slice).

Model slicing, as introduced in [8], is a model querying technique, rooted in the classical definition of program slicing, but extends it to UML class models.

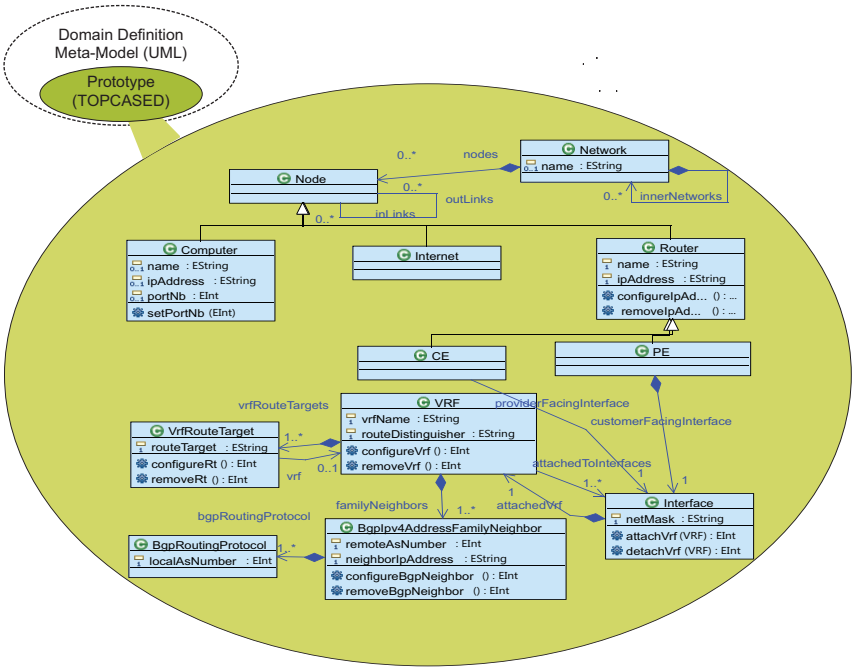

Fig. 2. Abstract Syntax of SGTSML

Program slicing, as defined by [9], applies a slicing criteria on a program to compute a slice (i.e.; a subset of the source code). Model slices are as well defined using a slicing criterion that is specified with predicates over the model's features. Consequently, model querying and model slicing in particular constitute a good starting point for our approach. However, because we will change elements in the output model (e.g.; for hierarchy shrinkage we will change the parent class in a generalization relation), we need mechanisms more powerful than just querying, we need model transformations.

As we mentioned in Sect. 1, we construct the DDMM by eliminating classes and shrinking inheritance hierarchies from NALs expressed as UML class models. We think that the most significant reductions will be due to hierarchy shrinkage. Therefore, we are particularly interested in hierarchy shrinkage methods. A more focused technique of program slicing is the class hierarchy slicing. An algorithm for slicing class hierarchies in C++ programs is described in [10]. This algorithm eliminates from an C++ class hierarchy the data and function members, classes and inheritance relations that are unnecessary for ensuring that the semantics of a program P that uses the hierarchy is maintained. However, this type of algorithm is *context-sensitive*, as it needs the program P that uses the hierarchy. The DDMM that we build is intrinsically *context-free*, as it has no knowledge and should not depend on the future models that will be defined using it. Therefore, such an approach is not suitable for us.

4 Enlarging the Domain Definition Metamodel

As we argued in Sect. 1, the DDMM is central for our approach. We started by defining a simple prototype, presented in Sect. 2. In order to enlarge the DDMM, we considered using domain analysis methods such as Family-oriented Abstractions Specification and Translation [11] or Organization Domain Modeling version 2 [12]. However, these methods require a lot of time. Moreover, the models defined using the language constructed around the DDMM should be easy to map towards existing models of network components (i.e.; NALs). Consequently, we decided to start from an NAL, specified as a large UML class model (Fig. 3), and define model querying (Sect. 3) rules such that the output model will correspond to the needs of service designers.

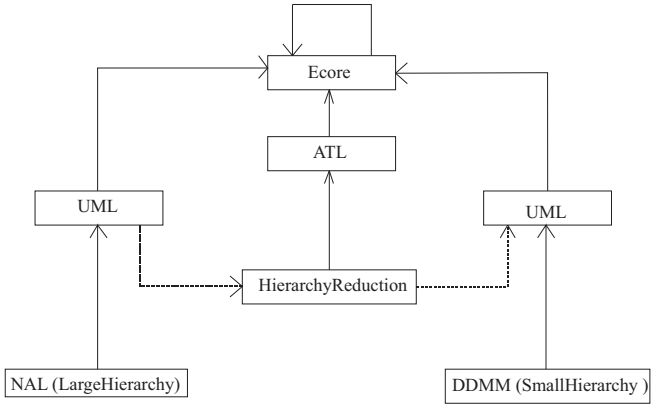

Fig. 3. Model Transformation

In Fig. 3, we exemplify the NAL through an UML class model called *LargeHierarchy*. LargeHierarchy conforms to its MM, which is UML. The UML MM can be written in several MM definition languages: MOF[4], Ecore[5], etc. We indicate here Ecore because we use Eclipse Modeling Tools[6] as toolkit and, consequently, the UML MM written in Ecore[7]. The output MM is also UML, and we call the output model *SmallHierarchy*, which is an example of a DDMM. Because the transformation has the same metamodel (i.e.; UML) for input and output, it is an endogenous transformation [13]. The transformation rules are written in the module *HierarchyReduction*, in ATL.

[4] http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF

[5] http://download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org/eclipse/emf/ecore/package-summary.html#details

[6] http://www.eclipse.org/downloads/packages/eclipse-modeling-tools-includes-incubating-components/ganymedesr1

[7] http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.mdt/org.eclipse.uml2/plugins/org.eclipse.uml2.uml/model/UML.ecore?root=Modeling_Project&view=log

The example we chose to illustrate the NAL, *LargeHierarchy*, is presented in Fig. 4. On one hand, an NAL's components are entities with attributes but no methods. The most frequent types of relation between these components are *association* and *generalization*. On the other hand, the service designers describe a service as a chain of calls to entities named *capabilities*, much as calls to functions. Therefore, a preliminary operation of mapping the capabilities (some hundreds) on the entities of the NAL (some tens of thousands) is done manually. This results in some of the NAL's classes having methods. They are represented in *LargeHierarchy* by the classes *B*, *C*, *E*, *F*, *G*, *H*. These classes will be part of the DDMM. We call this type of NAL classes, that after slicing appear in the DDMM, *generators*.

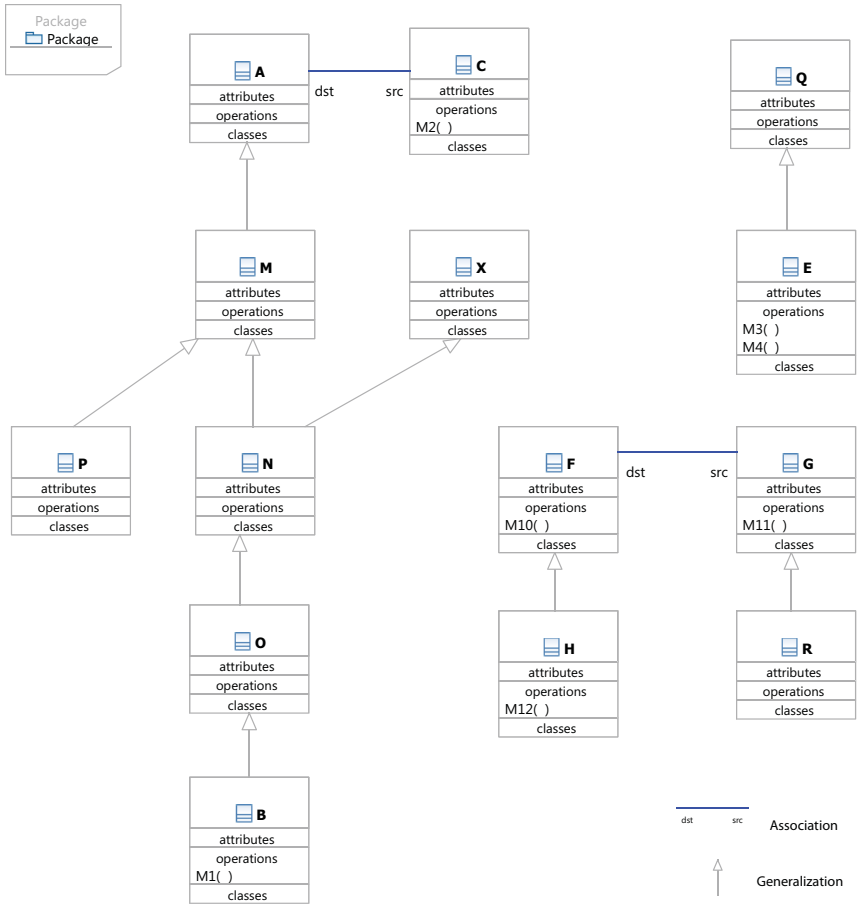

Fig. 4. Example of NAL: Large Hierarchy

The model transformation rules are presented, written in natural language, in Sect. 4.1 and written in ATL, in Sect. 4.2. The output model resulted from applying the model transformation rules on *LargeHierarchy* is presented in Fig. 5. We call it *SmallHierarchy* and it constitutes an example of DDMM. One can observe that all classes from *LargeHierarchy* that have at least one method (i.e.; B, C, E, F, G, H) exist in *SmallHierarchy* too. The initial relations between them (e.g.; the association relation from F and G and the generalization relation between F and H) also appear in the output model. The most interesting relation in *SmallHierarchy* is the association from classes B and C, resulted from the shrinkage of the initial hierarchy between A and B.

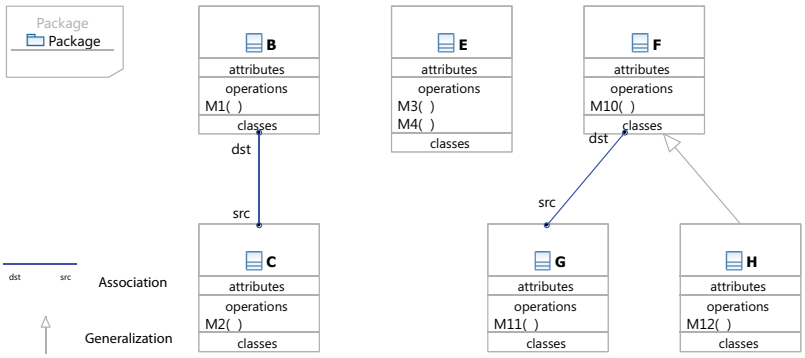

Fig. 5. Example of DDMM: Small Hierarchy

4.1 Model Transformation Rules

Our main idea for querying an NAL is to start from a set of initial generators (i.e.; the classes that have at least one method) and select as generators other classes that are related to the initial generators in a way that is relevant for service designers. The types of relation between classes from NAL, that we consider important for service designers, are association and generalization.

The rule to select the set of initial generators, in natural language:

1. Select all classes from NAL that have at least one method.

The rules for selecting **associations**, in natural language:

1. Select all direct associations from NAL that relate two generator classes.
2. We define the notion of *least derived generator* (ldSAG) as the generator which, in a hierarchy that contain generators in a generalization relation, is the highest in the hierarchy. Move association relations down in the hierarchy (i.e.; towards the more derived classes) to the least derived generator.

The rules for selecting **generalizations**, in natural language:

1. Select all direct generalizations from NAL that relate two generator classes.
2. Select all generators implementing an abstract generator.

4.2 Model Transformation Rules in ATL

In this section we present the rules for model transformation, written in ATL.

The rule to select the set of initial generators:

```
1  module HierarchyReductionUML; -- Module Template
2  create SmallHierarchyUML : UML from LargeHierarchyUML :
       UML;
3
4  rule Package {
5    from
6      ps : UML!Package
7    to
8      pt : UML!Package(
9        name <- ps.name)
10 }
11
12 rule Class {
13   from
14     cs : UML!Class(
15       cs.ownedOperation->notEmpty())
16   to
17     ct : UML!Class(
18       name <- cs.name,
19       package <- cs.package,
20       ownedOperation <- operationLst
21       ),
22       operationLst : distinct UML!Operation foreach
23         (oper in cs.ownedOperation.asSequence())(
24           name <- oper.name)
25 }
```

The rules related to **association** are *DirectAssociation* and *moveAssocDown-Hierarchy*:

```
1  rule DirectAssociation {
2    from
3      as : UML!Association(
4        thisModule.allSags->includes(as.memberEnd.
             asSequence()->first().type) and
5        thisModule.allSags->includes(as.memberEnd.
             asSequence()->last().type))
```

```
6     to
7       at : UML!Association(
8         name<-as.name,
9         package <- as.package,
10        memberEnd<-memberLst
11        ),
12      memberLst : distinct UML!Property foreach (asMember
            in as.memberEnd.asSequence())(
13          name<-asMember.name,
14          type<-asMember.type)
15  }
```

The rule *DirectAssociation* uses the attribute *allSags*:

```
1  --attributes
2  helper def : allSags : Set(UML!Class) =
3    let allClasses : Set(UML!Class) = UML!Class.
         allInstances() in
4    allClasses->select(i|i.ownedOperation->notEmpty());
```

```
1  rule moveAssocDownHierarchy{
2    --moves an association
3    --does NOT promote an element to an SAG
4    from
5      ps : UML!Property(
6        --the source element should have its participant in
             a hierarchy
7        if (ps.ldSagToMoveAssocDownHierarchyTo = '')
8        --the downHierarchy should have at least one SAG
9        --(to simplify the problem, I consider here only
             the ldSAGs)
10       then false
11       else
12         if (ps.doesDownHierarchyContainSeveralLdSags(ps.
               ldSagToMoveAssocDownHierarchyTo))
13         --the hierarchy should have only one ldSAG
14         --(it can have a SAG on only one branch)
15         then false
16         else true
17         endif
18       endif)
19   using{
20       targetLdSag : UML!Class = ps.
             ldSagToMoveAssocDownHierarchyTo;}
21   to
22     at : UML!Association(
```

```
23        --in the target model, we construct an association
              between one initial class and,
24        --instead of the other class, from the hierarchy,
              the ldSAG from its downHierarchy
25        name<-ps.association.name,
26        package <- ps.association.package,
27        memberEnd<-memberLst
28        ),
29        memberLst : distinct UML!Property
30          foreach (asMember in ps.association.memberEnd.
                asSequence())(
31            name <- asMember.name,
32            type <-    if asMember=ps
33                    then asMember.type
34                    else targetLdSag
35                    endif)
36  }
```

The rule *moveAssocDownHierarchy* exemplifies the fact that we go beyond model slicing. Not only that we select elements and relations from the input model, but we also change attributes on some of these relations (e.g.; the type of the *memberEnd* of the association is changed to the *targetLdSag*). This rule uses the helpers *ldSagToMoveAssocDownHierarchyTo*, *doesDownHierarchyContainSeveralLdSags*:

```
1   helper context UML!Property def :
        ldSagToMoveAssocDownHierarchyTo : UML!Class =
2     let otherMemberEnd : UML!Property =
3       if (self.association.memberEnd->asSequence()->first()
            = self)
4       then self.association.memberEnd->asSequence()->last()
5       else self.association.memberEnd->asSequence()->first
            ()
6       endif
7       in
8     thisModule.ldSags->iterate(ldSag; goodLdSag : UML!Class
          = '' |
9       if (ldSag.upperHierarchy->includes(otherMemberEnd.
            type))
10      then ldSag
11      else goodLdSag
12      endif);
13
14  helper context UML!Property def :
        doesDownHierarchyContainSeveralLdSags(firstLdSag : UML
        !Class) : Boolean =
```

```
15    let otherMemberEnd : UML!Property =
16      if (self.association.memberEnd->asSequence()->first()
            = self)
17      then self.association.memberEnd->asSequence()->last()
18      else self.association.memberEnd->asSequence()->first
            ()
19      endif
20      in
21    thisModule.ldSags->iterate(ldSag; has : Boolean = false
          |
22      if (ldSag <> firstLdSag)
23      then
24        if (ldSag.upperHierarchy->includes(otherMemberEnd.
              type))
25        then true
26        else has
27        endif
28      else
29        has
30      endif);
```

The helper *doesDownHierarchyContainSeveralLdSags* uses the attributes *ldSags* and *upperHierarchy*, the latter using at its turn the helpers *ancestors* and *parents* to navigate through the hierarchy:

```
1  helper def : ldSags : Set(UML!Class) =
2    let sags : Set(UML!Class) = thisModule.allSags
3    in sags->select(sag | sag.upperHierarchy->iterate(
4      i; notExists : Boolean = true |
5      if (sags->includes(i))
6      then notExists = false
7      else notExists
8      endif));
9      --sag.upperHierarchy->excludesAll(sags) );
10
11 helper context UML!Class def : upperHierarchy : Set(UML!
       Class) =
12   self.ancestors();
13
14 helper context UML!Class def : ancestors() : Set(UML!
       Class) =
15   let pars : Set(UML!Class) = self.parents() in
16   pars->union(
17     pars->iterate(parent; ancest : Set(UML!Class) = Set{}
           |
18     ancest->union(parent.ancestors())));
```

```
19
20 helper context UML!Class def : parents() : Set(UML!Class)
       =
21   let allGens : Set(UML!Class) = self.generalization in
22   allGens->iterate(gen; par : Set(UML!Class) = Set{} |
       par->union(Set{gen.general}));
```

The rules related to **generalization** are *DirectGeneralization* and *markAsSag*:

```
1  rule DirectGeneralization {
2    from
3      gs : UML!Generalization(
4        gs.general.ownedOperation->notEmpty() and gs.
             specific.ownedOperation->notEmpty())
5    to
6      gt : UML!Generalization(
7        general<-gs.general,
8        specific<-gs.specific)
9  }
```

```
1  rule markAsSag{
2    --promotes an element to an SAG
3    --updates the allSAG and ldSAG lists
4    --creates an association to the new SAG
5    --creates generalizations to the new SAG (if necessary)
6    from
7      ps : UML!Property(
8        if (ps.ldSagToMoveAssocDownHierarchyTo = '')
9        then false
10       else
11         if (ps.doesDownHierarchyContainSeveralLdSags(ps.
               ldSagToMoveAssocDownHierarchyTo))
12         then true
13         else false
14         endif
15       endif)
16   using{
17       otherMemberEnd : UML!Property =
18         if (ps.association.memberEnd->asSequence()->first
               () = ps)
19         then ps.association.memberEnd->asSequence()->last
               ()
20         else ps.association.memberEnd->asSequence()->
               first()
21         endif;
```

```
22        futureLdSag : UML!Class = otherMemberEnd.type;
23        auxLdSags : Set(UML!Class) = thisModule.ldSags;}
24    to
25      ct : UML!Class(
26        name <- futureLdSag.name,
27        package <- ps.association.package
28        ),
29      at : UML!Association(
30        name<-ps.association.name,
31        package <- ps.association.package,
32        memberEnd<-memberLst
33        ),
34        memberLst : distinct UML!Property
35          foreach (asMember in ps.association.memberEnd.
                asSequence())(
36            name <- asMember.name,
37            type <- asMember.type)
38    do{
39      thisModule.allSags <- thisModule.allSags->including(
            futureLdSag);
40      thisModule.ldSags <- thisModule.ldSags->including(
            futureLdSag);
41      for (ldSag in thisModule.ldSags){
42        if (ldSag.upperHierarchy->includes(futureLdSag)){
43          auxLdSags<-thisModule.ldSags.excluding(ldSag);
44          thisModule.createGeneralization(ldSag,
                futureLdSag);
45        }
46      }
47      thisModule.ldSags<-auxLdSags;
48    }
49  }
```

The rule *markAsSag* uses the rule *createGeneralization*:

```
1  rule createGeneralization(de : UML!Class, a : UML!Class){
2    to
3      gt : UML!Generalization(
4        general<-a,
5        specific<-de)
6  }
```

4.3 Preliminary performance results

In order to have an idea of the performance of the transformation, we did some preliminary tests. We used as machine a Dell Latitude E4300, with an Intel Core2 Duo CPU P9300 @ 2.26GHz 1.58GHz, 3.45Go RAM, with Microsoft XP SP3. As input model we used the model presented in Fig. 4, which we duplicated several times to obtain a bigger model. Table 1 shows on each line a model with increasing dimensions. The 'Factor' column represents the number of times the initial model has been duplicated. The column 'File' represents the dimension of the input model file, in bytes. The columns 'Classes', 'Associations' and 'Generalizations' represent the number of classes, associations and generalizations respectively contained by each model. The column 'Execution time' represents the execution time, in seconds, of the transformation rules applied on each model respectively. To measure the execution time we used the Eclipse facilities (Run→Run Configurations, Advanced tab, and select 'Run mode only: print execution times to console: 1) transformation only, and 2) total (including model loading and saving)'; we mention that there was only one time printed at the console). The result of under 3 minutes for a model containing approximately 20.000 entities encourages us to think that, when applied on industrial-scale NALs, the transformation will have satisfactory execution times.

Crt.	Factor	File (B)	Classes	Associations	Generalizations	Execution time (s)
1	1	6,105	14	2	8	0.093
2	8	42,400	104	16	64	0.312
3	64	306,348	832	128	256	6.5
4	1,024	2,120,162	14,336	2,048	8,192	161.531

Table 1. Performance results

5 Lessons Learned

High level of abstraction. We have found that using ATL to describe our model querying algorithm offers a high level of abstraction, especially when compared to hierarchy slicing algorithms like the one presented in [10], due to its declarative constructions (i.e.; the matching of model elements).

Expressive code. The code written to implement the algorithm is much more compact and expressive than if written in a general purpose programming language, like C++; this is a direct consequence of ATL being a DSL for model transformation.

Code modularization and change management. Rule definition provides a strong mechanism for code modularization (i.e.; a rule encodes by itself all the functionality) and change management (e.g.; adding new behavior to the algorithm is as simple as writing new rules).

Performance. The preliminary results we obtained encourage us to think of ATL as applicable to industrial-size models.

Tool support. ATL comes with a virtual machine, an editor with syntax highlighting and code completion for metamodel elements, a debugger. Although sufficiently mature to support development, these tools have missing features that would increase their efficiency (e.g.; adding a breakpoint has to be done from the outline view, there is no code completion for rules, attributes, helpers defined in the same module, no code completion for data types). Also, there are minor bugs (e.g.; the operation *excludesAll* on a collection does not work in the ATL version[8] we used - we had to find a workaround - see helper *ldSags*).

Functional programming style. The functional programming style (e.g.; used to specify the conditions for matching) may be difficult for many programmers to use. Moreover, this programming style produces complex and long expressions, hard to read and understand. Having the documentation of an element appear as a tool tip when hovering over it may be highly useful.

Factorization limits. When comparing the rules *moveAssocDownHierarchy* and *markAsSag*, one observes that the *from* parts are very similar. We have actually tried to write only one rule, but did not succeed. However, having different rules contributes to the modularity and readability of the code, as each addresses different functionality.

6 Conclusion and Future Work

In this work we were interested in defining a Domain Definition Metamodel (DDMM) for Telecommunications service definition by model querying and transforming large Network Abstraction Layers (NALs) expressed as UML models. We defined the querying and transformation rules in ATL, finding this approach well suited. In the future, we intend to measure the performance of our model transformation rules on industrial-scale NALs (tens of thousands of classes). We also plan to evaluate the DDMM against service designers and use their input to further enlarge and refine it.

References

1. Fahy, C., Davy, S., Boudjemil, Z., van der Meer, S., Loyola, J., Serrat, J., Strassner, J., Berl, A., de Meer, H., Macedo, D.: Towards an Information Model That Supports Service-Aware, Self-managing Virtual Resources. In: Proceedings of the 3rd IEEE international workshop on Modelling Autonomic Communications Environments, Springer (2008) 102–107
2. Barrett, K., Davy, S., Strassner, J., Jennings, B., van der Meer, S., Donnelly, W.: A Model Based Approach for Policy Tool Generation and Policy Analysis. In: Proceedings of the IEEE Global Information Infrastructure Symposium. (2007) 99–105

[8] org.eclipse.m2m.atl.engine.vm_2.0.0

3. Bezivin, J., Dupe, G., Jouault, F., Pitette, G., Rougui, J.: First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In: 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture. (2003)
4. Kurtev, I., Bezivin, J., Jouault, F., Valduriez, P.: Model-based DSL frameworks. In: OOPSLA '06:. (2006) 602–616
5. Chiprianov, V., Kermarrec, Y.: Model-based DSL Frameworks: A Simple Graphical Telecommunications Specific Modeling Language. In: Actes des 5 émes journées sur l'Ingénierie Dirigée par les Modéles. (2009)
6. Farail, P., Gaufillet, P., Canals, A., Le Camus, C., Sciamma, D., Michel, P., Cregut, X., Pantel, M.: The TOPCASED project: a Toolkit in Open source for Critical Aeronautic Systems Design. In: ERTS. (2006)
7. Features, C.: openArchitectureWare 4.2. Technical report, Eclipse (2007)
8. Kagdi, H., Maletic, J., Sutton, A.: Context-Free Slicing of UML Class Models. In: Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on. (2005) 635–638
9. Weiser, M.: Program Slicing. In: Proceedings of the 5th international conference on Software engineering, IEEE Press Piscataway, NJ, USA (1981) 439–449
10. Tip, F., Choi, J., Field, J., Ramalingam, G.: Slicing class hierarchies in C++. ACM SIGPLAN Notices **31**(10) (1996) 179–197
11. Coplien, J., Hoffman, D., Weiss, D.: Commonality and Variability in Software Engineering. IEEE Softw. **15** (1998) 37–45
12. Simos, M., Anthony, J.: Weaving the Model Web: A Multi-Modeling Approach to Concepts and Features in Domain Engineering. In: ICSR '98. (1998)
13. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science **152** (2006) 125–142